Rust Book - 3.2: Data Types

#rust Data Types

Every value in Rust is of a certain*data type*, which tells Rust what kind of data is being specified so it knows how to work with that data.

Rust is a **statically typed** language, which means that it must know the types of all variables at compile time. The compiler can usually infer what type we want to use based on the value and how we use it.

Type annotation:

```
let guess = “42”.parse().expect(“Not a number!”);
```

vs.

```
let guess: u32 = “42”.parse().expect(“Not a number!”);
```

Scalar Types

Scalar Types

A*scalar*type represents a single value. Rust has four primary scalar types: integers, floating-point numbers, Booleans, and characters. You may recognize these from other programming languages. Let’s jump into how they work in Rust.

The Character Type

the **char** literal is specified with single quotes, as opposed to string literals, which use double quotes.

Compound Types

Compound Types

Compound types can group multiple values into one type. Rust has two primitive compound types: tuples and arrays.

The Tuple Type

A tuple is a general way of grouping together some number of other values with a variety of types into one compound type. Tuples have a fixed length: once declared, they cannot

grow or shrink in size.

```
fn main() {
    let tup = (500, 6.4, 1);
    let (x, y, z) = tup;
}
```

This program first creates a tuple and binds it to the variable **tup**. It then uses a pattern with **let** to take tup and turn it into three separate variables ,x,y, and z. This is called*destructuring*, because it breaks the single tuple into three parts.

The Array Type

Another way to have a collection of multiple values is with an*array*. Unlike a tuple, every element of an array must have the same type. Arrays in Rust are different from arrays in some other languages because arrays in Rust have a fixed length, like tuples.

If you're unsure whether to use an array or a vector, you should probably use a vector.

```
fn main() {
    let a = [1, 2, 3, 4, 5];
    let index = 10;
    let element = a[index];
}
```

… program resulted in a runtimeerror and didn't exit successfully. When you attempt to access an element using indexing, Rust will check that the index you've specified is less than the array length. If the index is greater than the length, Rust will panic.

This is the first example of Rust's safety principles in action. In many low-level languages, this kind of check is not done, and when you provide an incorrect index, invalid memory can be accessed. Rust protects you against this kind of error by immediately exiting instead of allowing the memory access and continuing.

…